Last update: June 1, 2017

NTT

Innovative R&D by NTT

Parallelizing CI using Docker Swarm-Mode

Akihiro Suda <suda.akihiro@lab.ntt.co.jp>
NTT Software Innovation Center

github.com/AkihiroSuda

- **Software Engineer at NTT Corporation**

- **Several talks at FLOSS community**
  - FOSDEM 2016
  - ApacheCon Core North America 2016, etc.

- **~~Docker~~ Moby Project core maintainer**

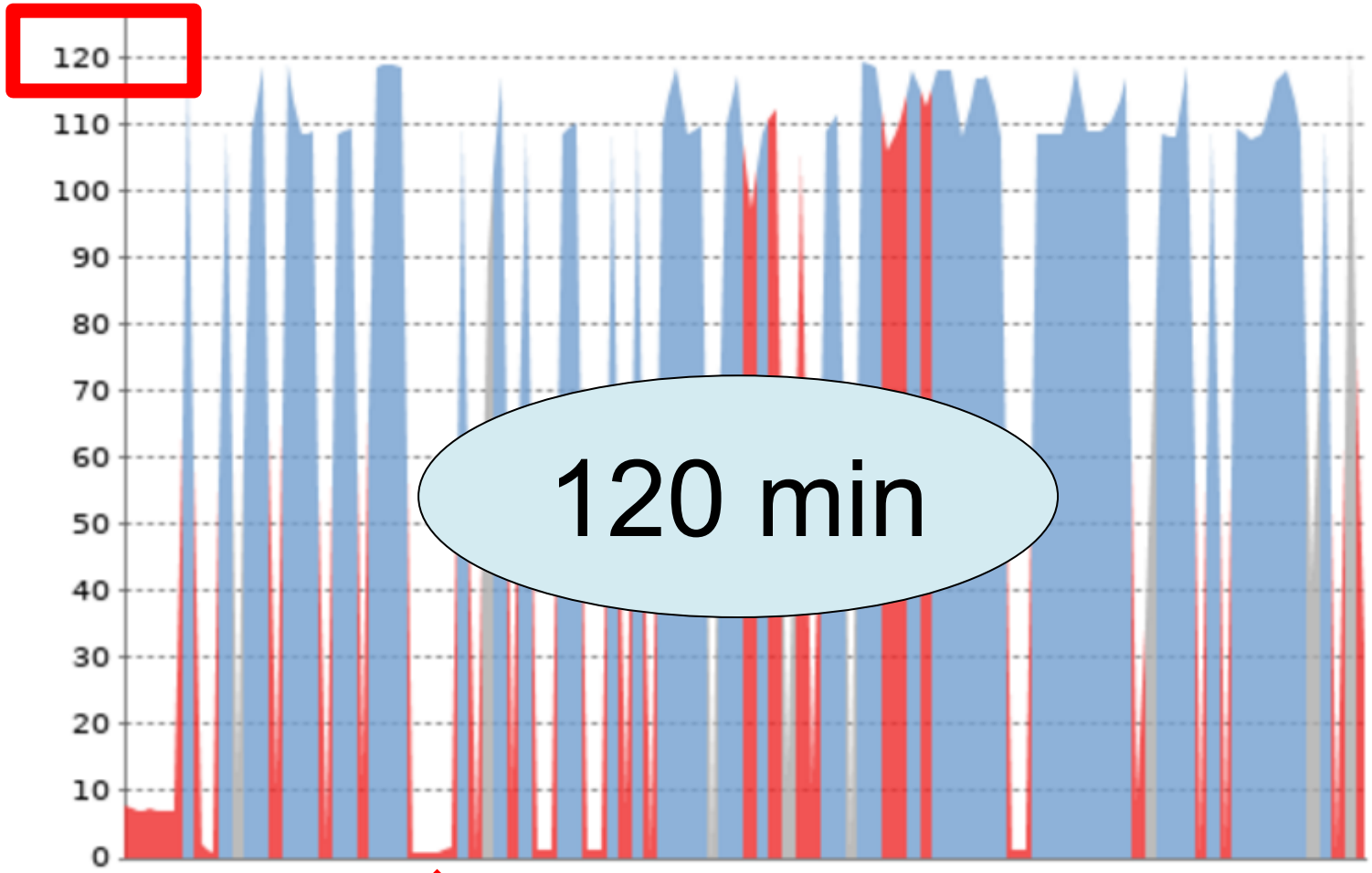
Docker project transited into Moby Project (April, 2017).
Now Docker products are "downstreams" of Moby Project.

docker : moby project ≒ : Fedora

A problem in Docker/Moby project: CI is slow

capture: March 3, 2017

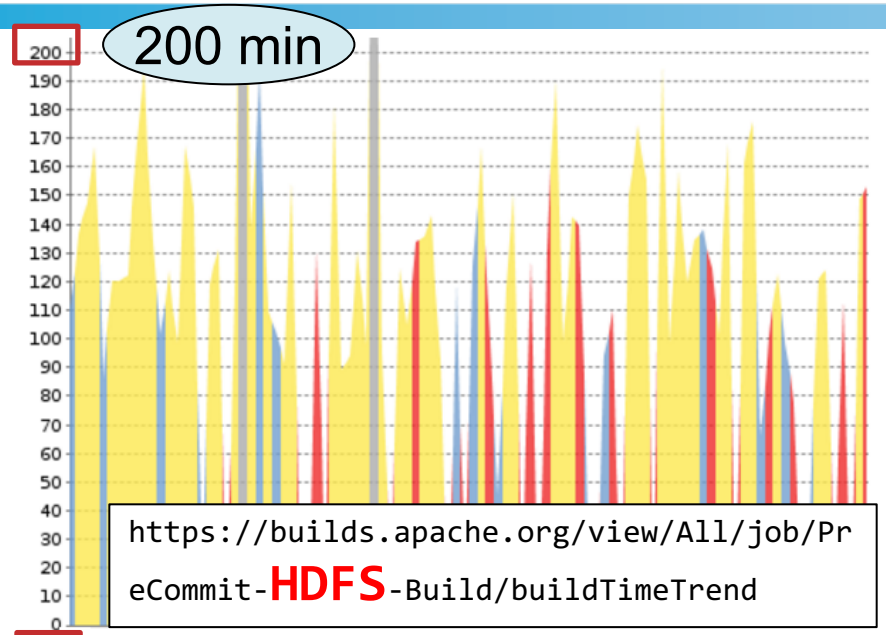
https://jenkins.dockerproject.org/job/Docker-PRs/buildTimeTrend

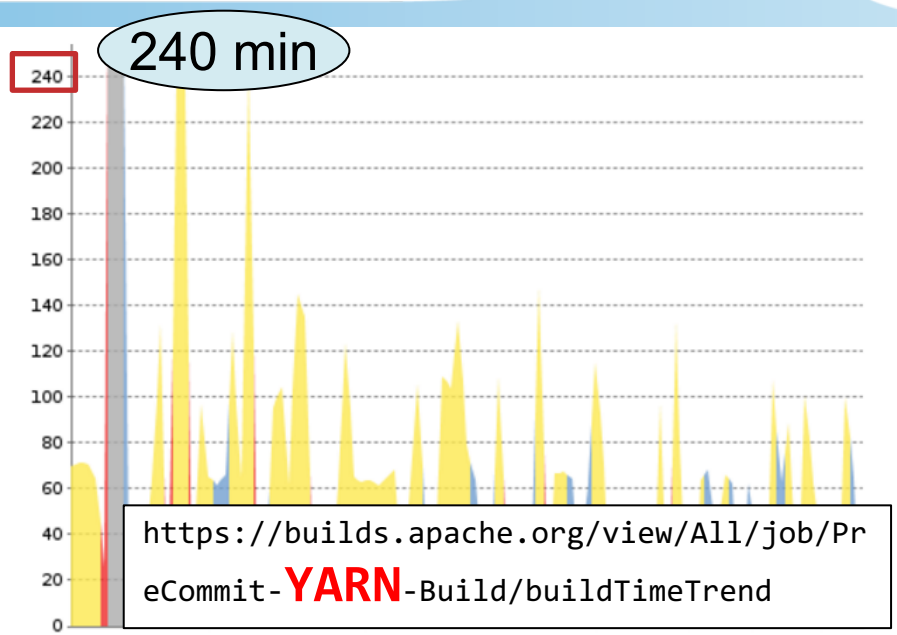


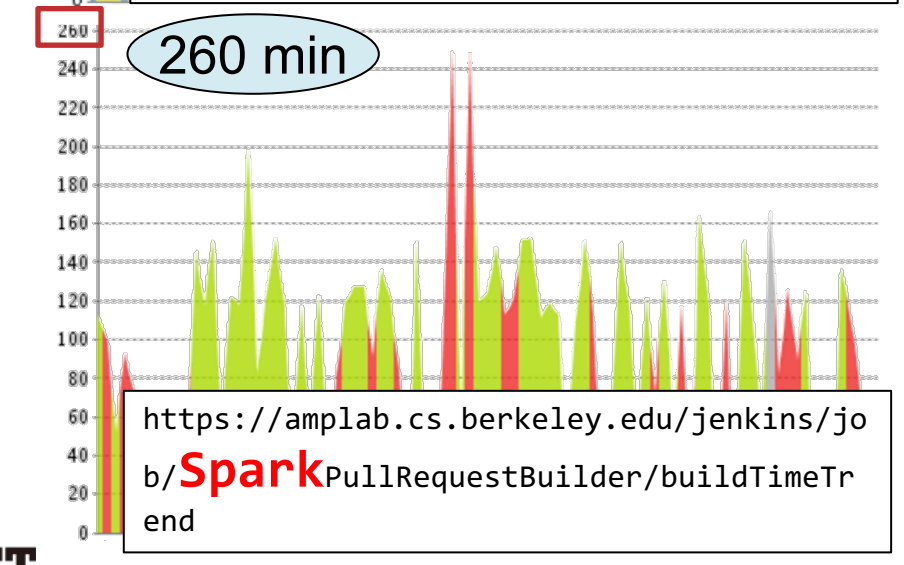
red valley = test failed immediately ☹

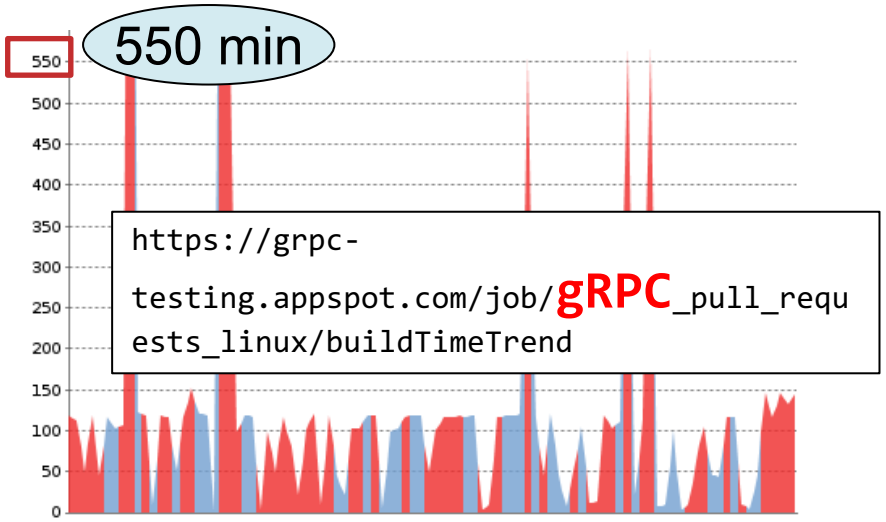
How about other FLOSS projects?

capture: March 3, 2017









(for ease of visualization, picked up some projects that use Jenkins from various categories)

Why slow CI matters?

- **Blocker for reviewing/merging patches**
- **Discourages developers from writing tests**
- **Discourages developers from enabling additional testing features**
  - e.g. `go test -race` (race detector) is 2-20x slower

source: https://golang.org/doc/articles/race_detector.html

Why slow CI matters?

Can result in
poor implementation quality &
slow development cycle 😥

Solution: Parallelize CI ?

Go

```
$ go test -parallel N ...
```

Java (Maven)

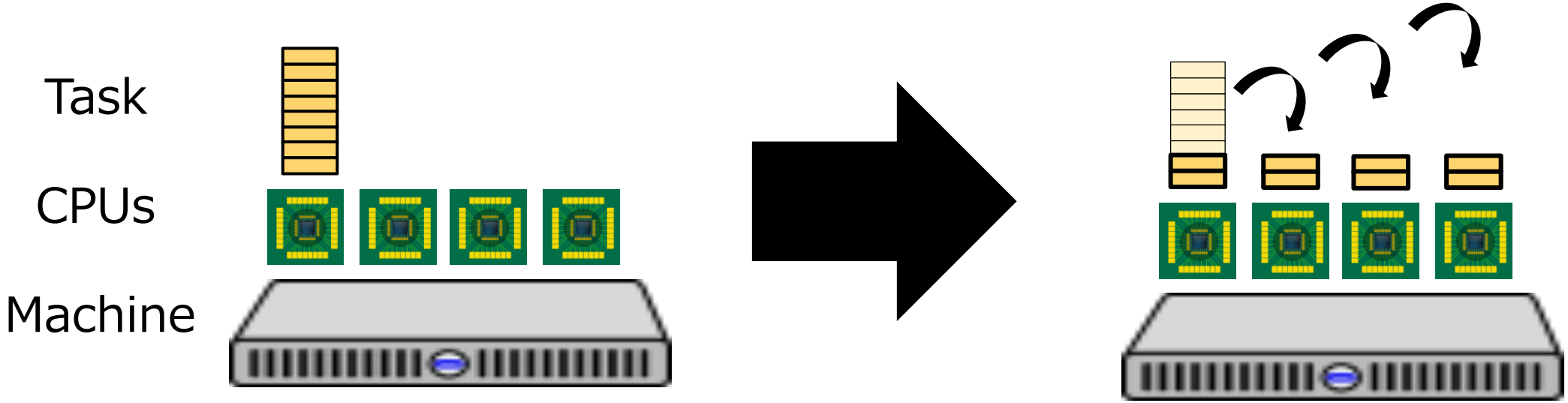
```
$ mvn --threads N ...
```

Python (nose)

```
$ nosetests --processes N ...
```

General

```
$ parallel --max-procs N ...
```



Solution: Parallelize CI ?

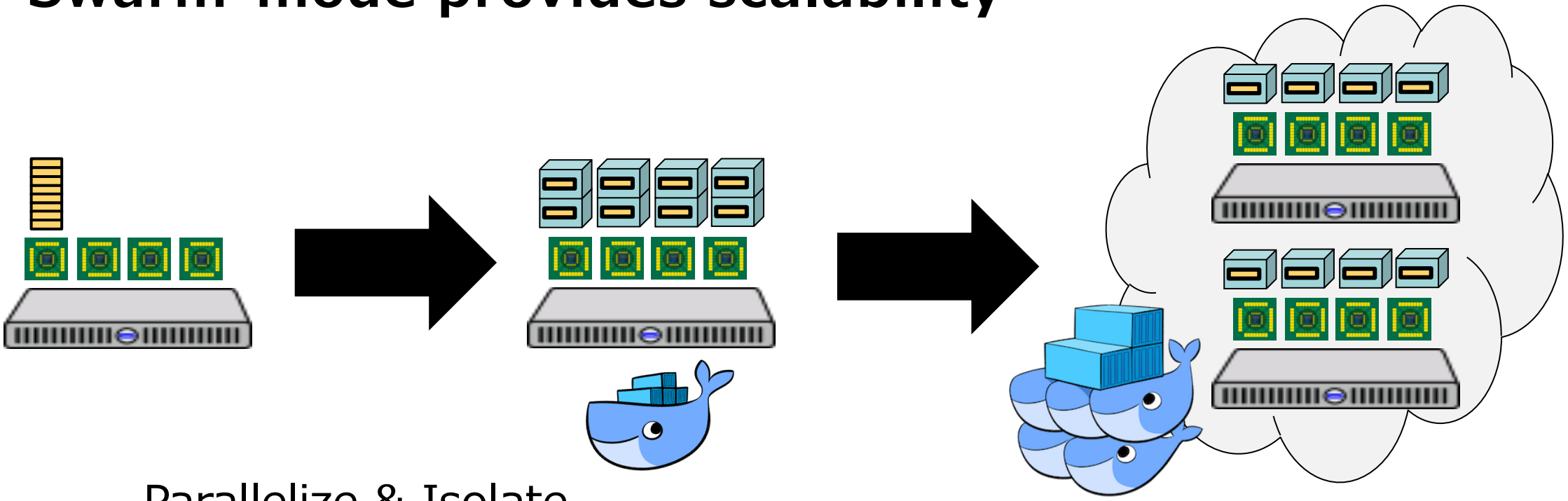
But just doing parallelization is not enough

- **No isolation**
  - Concurrent test tasks may race for certain shared resources (e.g. files under `/tmp`, TCP port, ...)

- **Poor scalability**
  - CPU/RAM resource limitation
  - I/O parallelism limitation

Solution: Docker (in Swarm-mode)

- **Docker provides isolation**
- **Swarm-mode provides scalability**



Parallelize & Isolate
using Docker containers

✓ Isolation

Distribute across Swarm-mode

✓ Scalability

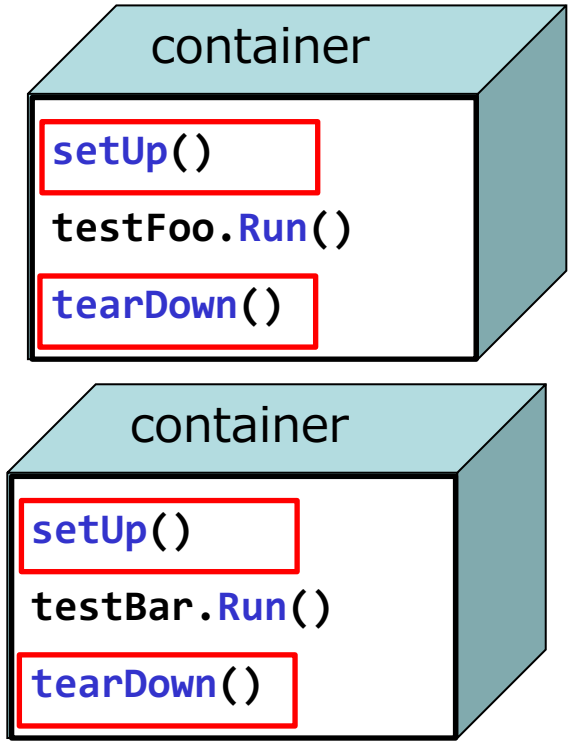
Challenge 1: Redundant setup/teardown

- For ideal isolation, each of the test functions should be encapsulated into independent containers

- But this is not optimal in actual due to setup/teardown code

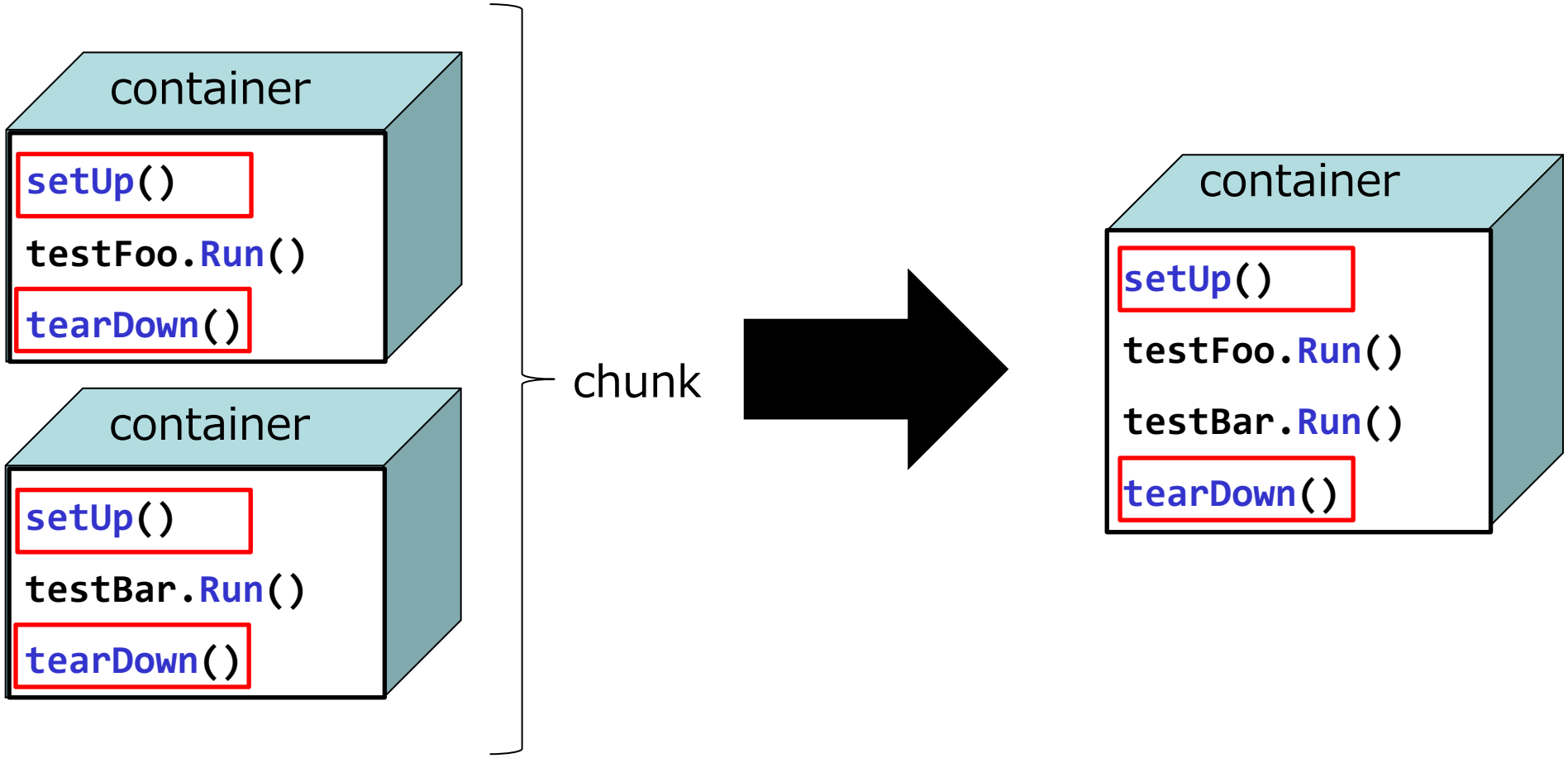
```
func TestMain(m *testing.M) {
    setup()
    m.Run()
    tearDown()
}
func TestFoo(t *testing.T)
func TestBar(t *testing.T)
```

redundantly executed



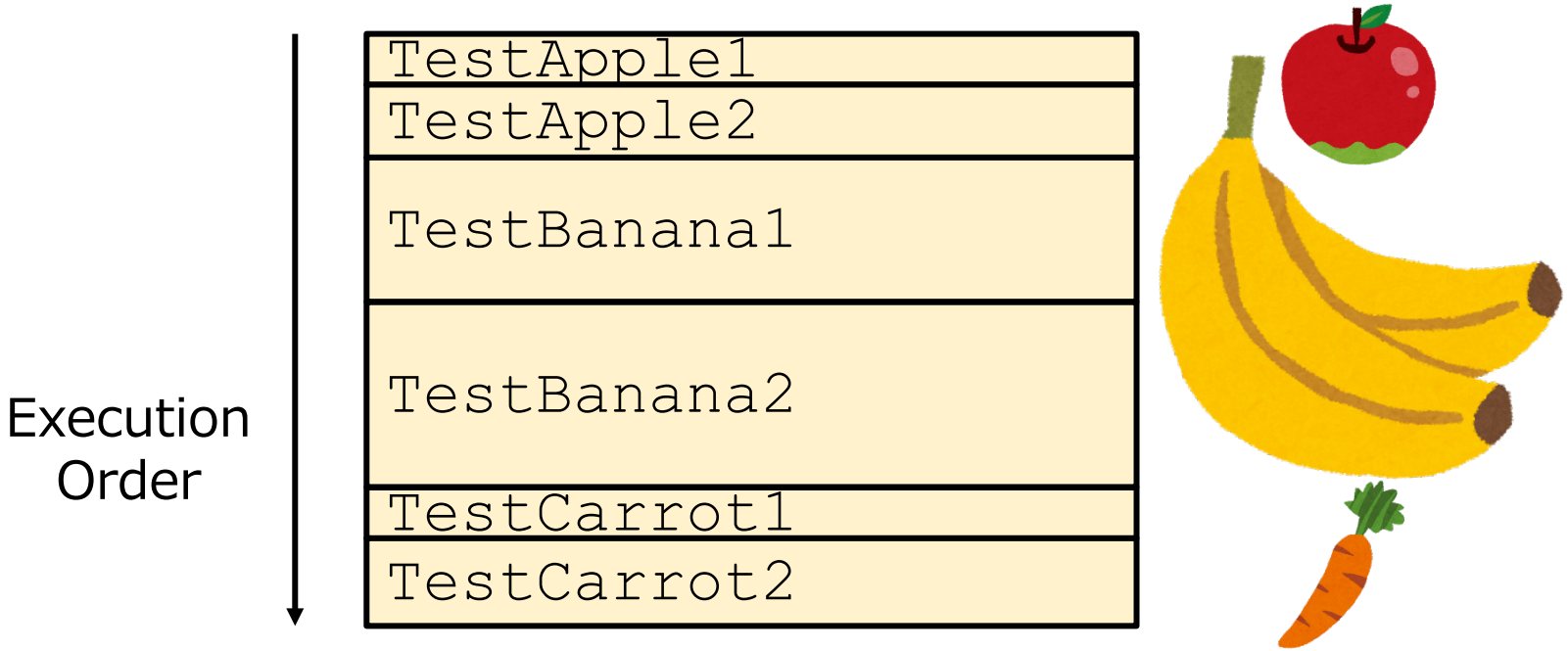
Optimization 1: Chunking

- **Solution: execute a chunk of multiple test functions sequentially in a single container**



Challenge 2: Makespan non-uniformity

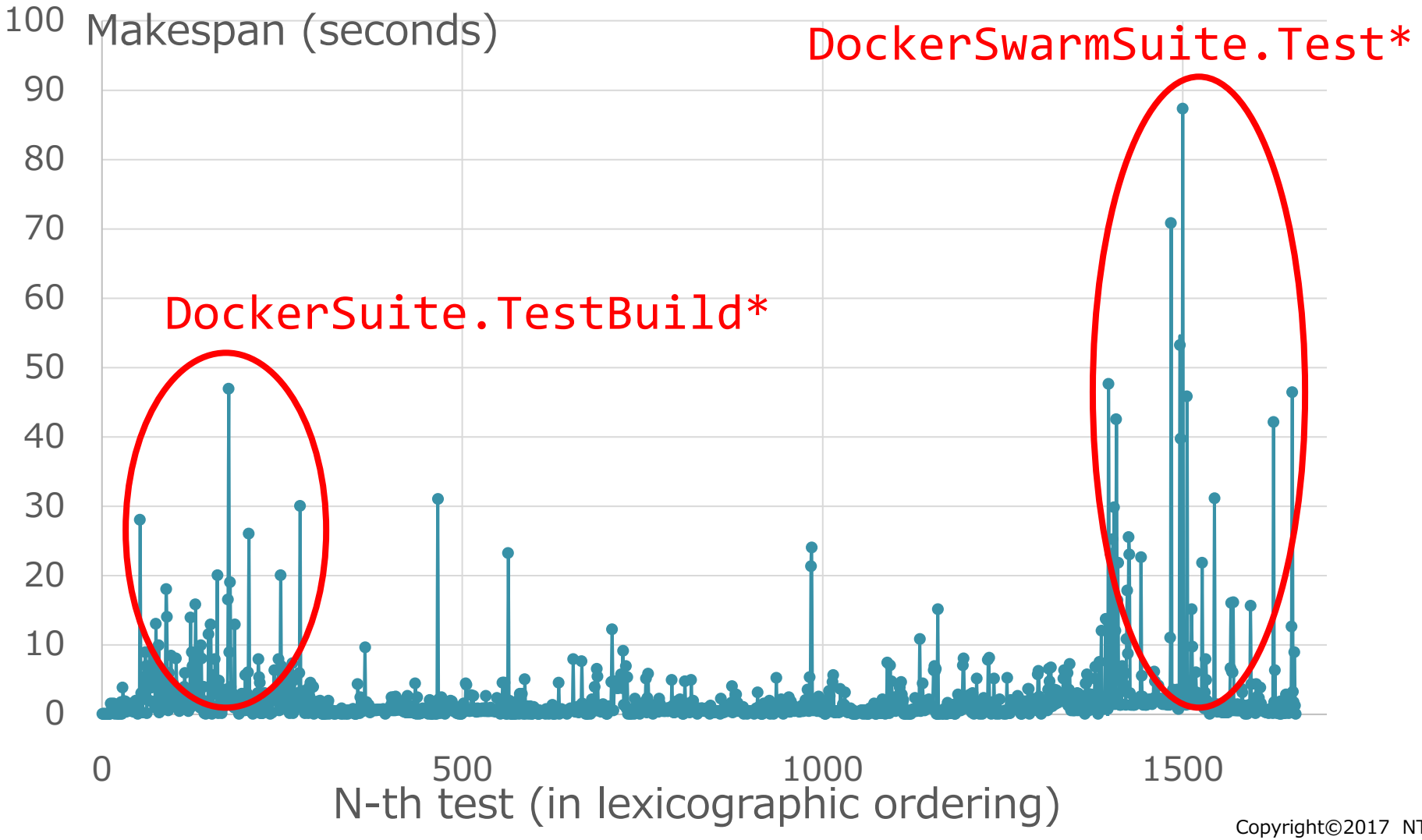
- **Test sequence is typically executed in lexicographic order (ABC…)**



- **Observation: long-running test functions concentrate on a certain portion**
  - because testing similar scenarios

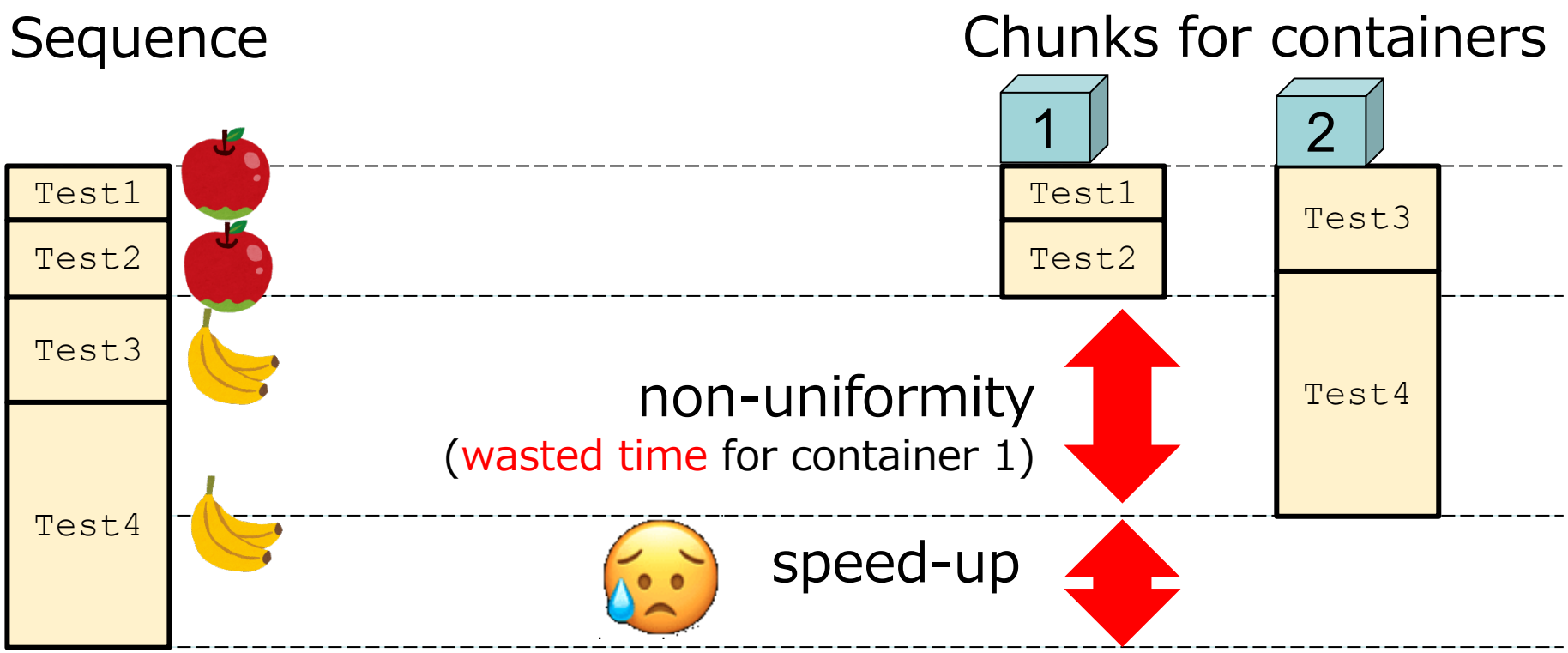
Challenge 2: Makespan non-uniformity

Example: Docker itself



Challenge 2: Makespan non-uniformity

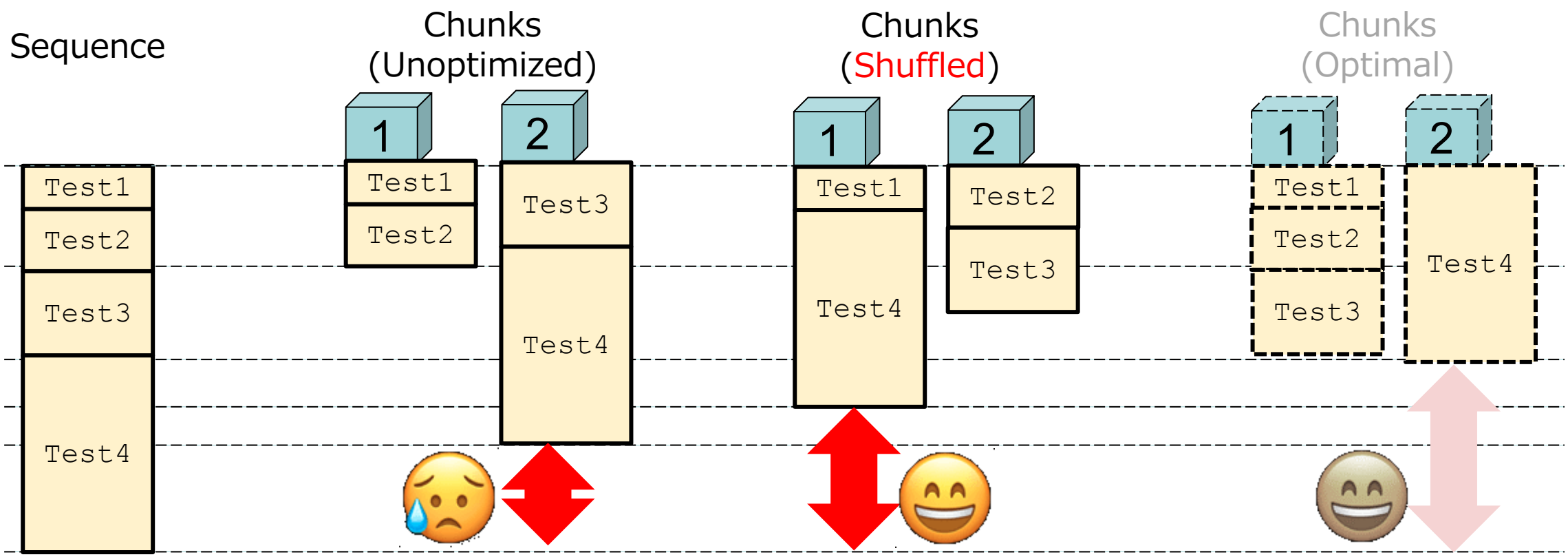
Makespans of the chunks are likely to result in non-uniformity



Optimization 2: Shuffling

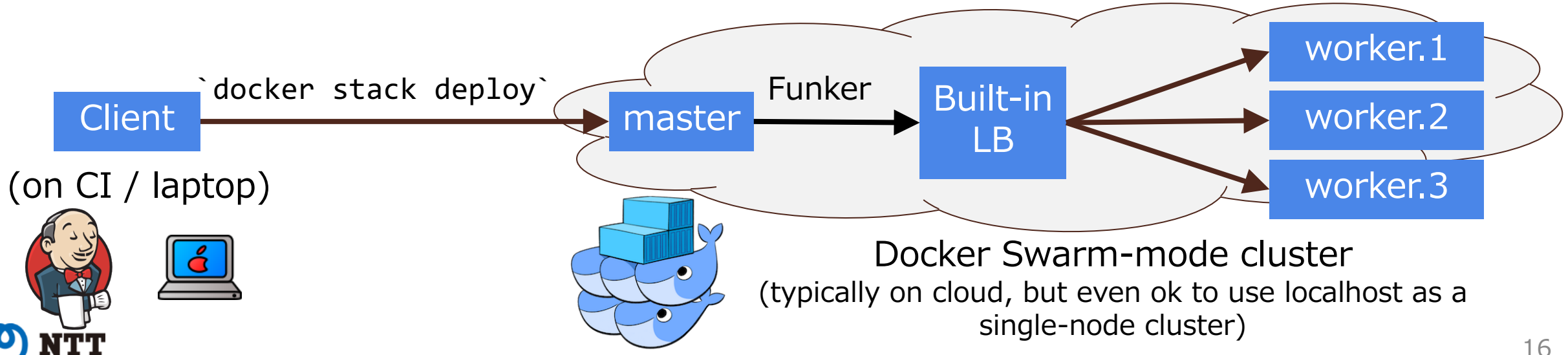
Solution: shuffle the chunks

- No guarantee for optimal schedule though



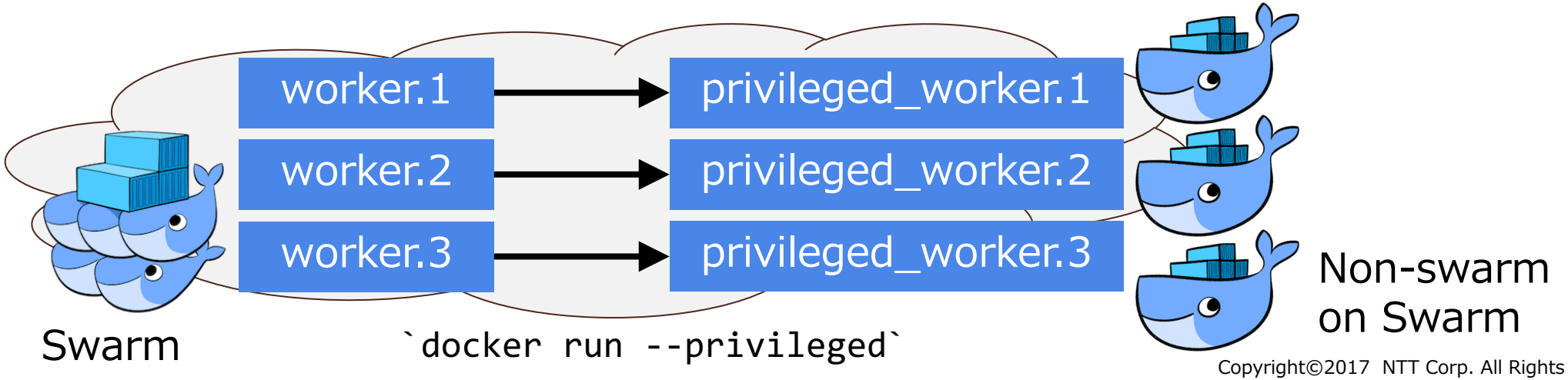
Implementation

- **RPC: Funker (github.com/bfirsh/funker)**
  - FaaS-like architecture
  - Workloads are automatically balanced via Docker's built-in LB
  - No explicit task queue; when a worker is busy, the master just retries
- **Deployment: `docker stack deploy` with Compose file**
- **Easily portable to other orchestrators as well (e.g. Kubernetes)**



Implementation (Docker-specific part)

- **Testing Docker itself requires `docker run --privileged`**
- **But Swarm-mode lacks support for `--privileged` at the moment**
  - moby/moby#24862
- **Workaround: Bind-mount `/var/run/docker.sock` into service containers, and execute `docker run --privileged` within them**

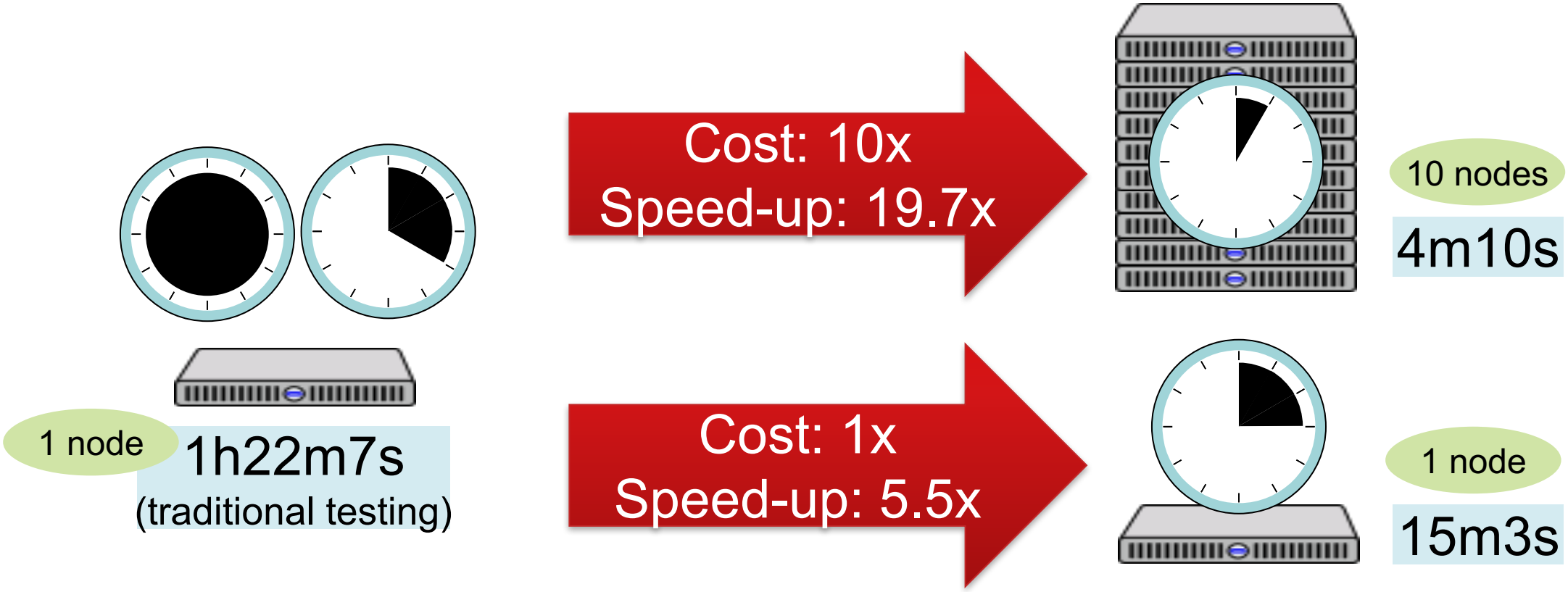


Experimental result

- **Evaluated my hack against the CI of Docker itself**
  - Of course, this hack can be applicable to CI of other software as well
- **Target: Docker 16.04-dev (git:`7fb83eb7`)**
  - Contains 1,648 test functions
- **Machine: Google Compute Engine `n1-standard-4` instances (4 vCPUS, 15GB RAM)**

Experimental result

- **20 times faster at maximum with 10 nodes**
- **But even effective with a single node!**



Detailed result

Containers running in parallel

Time: average of 5 runs, Graph driver: overlay2

Nodes	1 (Chunk size: 1648)	10 (165)	30 (55)	50 (33)	70 (24)
1	**1h22m7s** (=traditional)	**15m3s** 5.5x (BCR 5.5x)	N/A (more than 30m)		
2		12m7s	**10m12s** 8.1x (BCR 4.0x)	11m25s	13m57s
5		10m16s	6m18s	**5m46s** 14.2x (BCR 2.8x)	6m28s
10		8m26s	4m31s	**4m10s** 19.7x (BCR 2.0x)	4m20s

Best BCR (benefit-cost ratio)

Fastest configuration

What if no optimization techniques?

See the previous slide for the number of containers running in parallel

Nodes	Parallelize	Parallelize + Chunking	Parallelize + Chunking + Shuffling (= previous slide)
1	more than 30m	14m58s	15m3s
2		10m1s	10m12s
5		7m32s	5m46s
10		6m9s	4m10s

Significantly faster

1.5x faster

Scale-out vs Scale-up?

Scale-out wins

- Better I/O parallelism, mutex contention, etc.

	Nodes	Total vCPUs	Total RAM	Containers	Result
Scale out	10	40	150GB	50	4m10s
Scale up	1				19m17s

with both chunking and shuffling

The code is available!

PR (merged): docker/docker#29775

bash

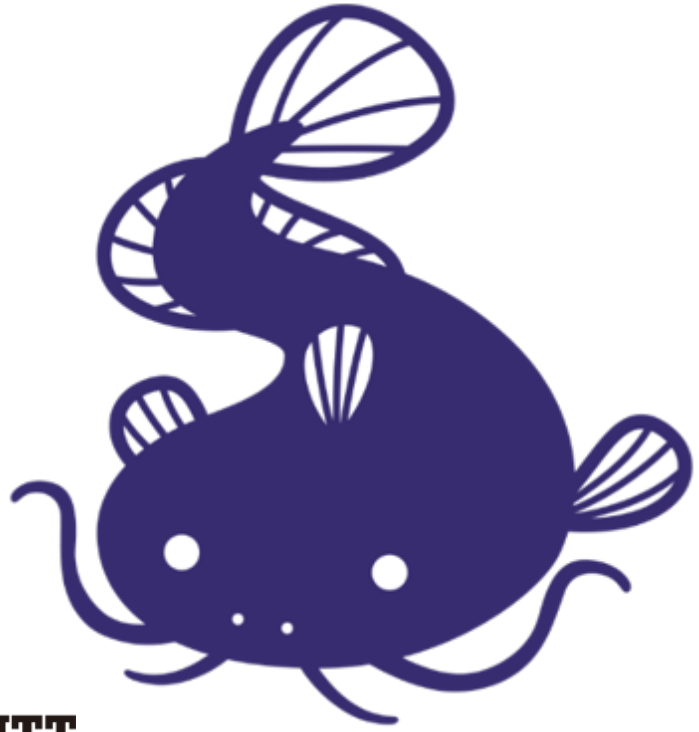
```
$ cd $GOPATH/src/github.com/docker/docker
$ make build-integration-cli-on-swarm
$ ./hack/integration-cli-on-swarm/integration-cli-on-swarm \
  -push-worker-image your-docker-registry/worker:latest \
  -replicas 50 \
  -shuffle
```

Is it applicable to other software as well?

Yes, generalized & simplified version available

(some Docker-specific hacks were eliminated)

github.com/osrg/namazu-swarm



"Namazu Swarm"

- Namazu (鯰) means a catfish in Japanese
- Our related project: github.com/osrg/namazu

  (A tool for reproducing flaky tests and injecting faults)
- *Un*related: www.namazu.org (text search engine)

Easy to get started

Just write a `Dockerfile` with two labels!

vi Dockerfile

```
FROM  your-project
ADD   tests.txt /

LABEL \
  net.osrg.namazu-swarm.v0.master.script="cat /tests.txt" \
  net.osrg.namazu-swarm.v0.worker.script="sh -e -x"
~
~
```

Emit all the test IDs (== commands, typically) to stdout

Read a chunk of test IDs from stdin, and execute them

Easy Integration with CI and Clouds

Namazu Swarm itself is tested on Travis

Travis CI

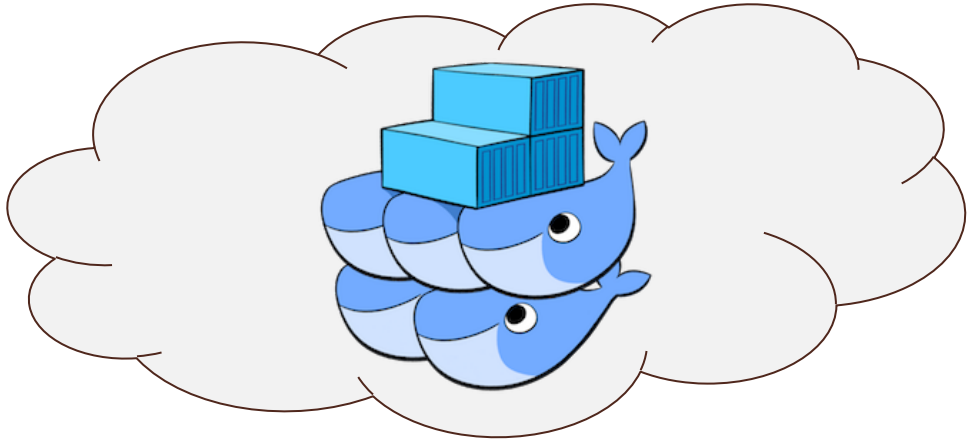
Circle CI

Jenkins

(Laptop)

Planned – Kubernetes
(e.g. GKE, ACS..)

Planned - ECS



Docker Swarm-mode cluster

(typically on cloud, but even ok to use localhost as a single-node cluster)

Experimental result

Apache ZooKeeper

Traditional	1 Node (10 containers)	2 Nodes (30 containers)	5 Nodes (50 containers)	10 Nodes (90 containers)
56m7s	19m34s	17m19s	9m50s	7m52s

2.9x faster (Cost: 1x)

7.1x faster (Cost: 10x)

Your own application

- Your report is welcome

Future work

- **Record and use the past execution history to optimize the schedule, rather than just shuffling**

- **Investigate deeply why scale-out wins**
  - related: moby/moby#33254

Future work

- **Mitigate extra overhead of pushing/pulling the image to/from the registry**
  - Can take a few minutes, depending on the network condition and the amount of the cached layers on each of the nodes
  - FILEgrain: github.com/AkihiroSuda/filegrain
    - My lazy-pull extension for OCI Image Spec
    - Experimental result for `java:8` image

Image format	Workload	Pull
Traditional Docker/OCI	Any	633 MB (100%)
FILEgrain	`sh`	4 MB (0.6%)
	`java -version`	87 MB (14%)
	`javac Hello.java`	136 MB (22%)

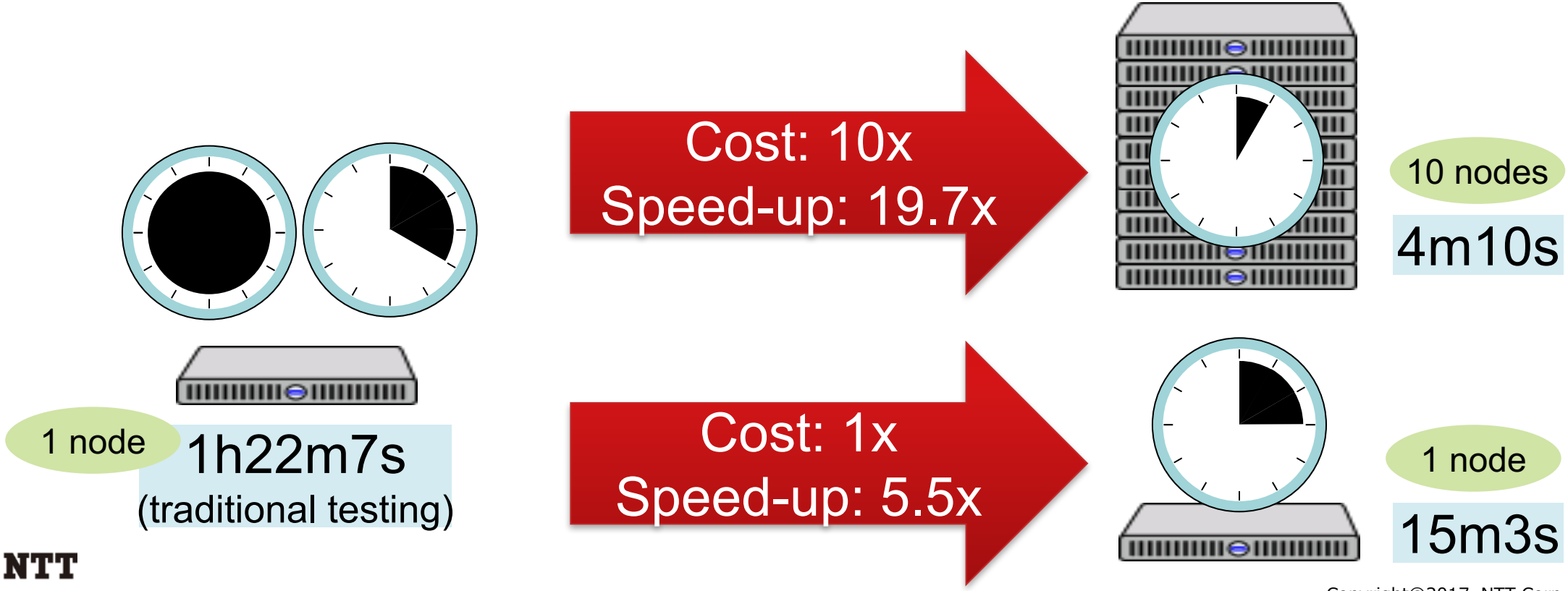
Still POC..
your contribution is welcome

  - P2P image distribution? e.g. IPFS

Recap

- **Docker Swarm-mode is effective for parallelizing CI jobs**

- **Introduced some techniques for optimal scheduling**



Recap

- You can easily apply the tool to your software as well
  github.com/osrg/namazu-swarm